Compressed Index for Dynamic Text

Wing-Kai Hon* Tak-Wah Lam* Kunihiko Sadakane†
Wing-Kin Sung‡ Siu-Ming Yiu*

Abstract

This paper investigates how to index a text which is subject to updates. The best solution in the literature [6] is based on suffix tree using $O(n \log n)$ bits of storage, where n is the length of the text. It supports finding all occurrences of a pattern P in $O(|P| + occ)$ time, where occ is the number of occurrences. Each text update consists of inserting or deleting a substring of length y and can be supported in $O(y + \sqrt{n})$ time. In this paper, we initiate the study of compressed index using only $O(n \log |\Sigma|)$ bits of space, where Σ denotes the alphabet. Our solution supports finding all occurrences of a pattern P in $O(|P| \log^2 n(\log^\epsilon n + \log |\Sigma|) + occ \log^{1+\epsilon} n)$ time, while insertion or deletion of a substring of length y can be done in $O((y + \sqrt{n}) \log^{2+\epsilon} n)$ amortized time, where $0 < \epsilon \leq 1$. The core part of our data structure is based on the recent work on Compressed Suffix Trees (CST) and Compressed Suffix Arrays (CSA).

1 Introduction

With the proliferation of DNA texts in recent years, there is an increasing interest in creating indexes for these texts so that pattern searching can be performed efficiently. As DNA texts are usually very long, traditional indexes like suffix trees and suffix arrays suffer from the problem of inadequate main memory. To further complicate the matter, these DNA texts are subject to frequent updates, as errors are common in the sequencing process. Therefore, a compressed (or more space-efficient) index for a text that allows efficient pattern searching and fast updating is highly desirable.

Before discussing the solution to the above *compressed dynamic text indexing* problem, let us have a quick review on the achievements of the traditional indexes. Let T be a text of length n, with characters drawn from an alphabet Σ. The suffix tree [14] is perhaps the most popular index for pattern searching when the text is *static*. The space requirement is $O(n \log n)$ bits, and it supports finding all occurrences of a pattern

*Department of Computer Science, The University of Hong Kong, Hong Kong, {wkhon,twlam,smyiu}@cs.hku.hk, phone: (852)-28578242, fax: (852)-25598447

†Department of Computer Science and Communication Engineering, Kyushu University, Japan, sada@csce.kyushu-u.ac.jp

‡School of Computing, National University of Singapore, Singapore, ksung@comp.nus.edu.sg

Proceedings of the Data Compression Conference (DCC'04)
1068-0314/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

P among the text T in an $O(|P| + occ)$ time, where occ is the number of occurrences. Another well-known index is the suffix array [13], which also occupies $O(n \log n)$ bits, and supports pattern searching of P in $O(|P| + \log n + occ)$ time.

When the text is subject to change, Ferragina and Grossi [6] proposed an *interval partitioning* scheme to exploit the *generalized suffix tree* [10] to give an index that occupies $O(n \log n)$ bits of space, and supports searching of P in $O(|P| + occ)$ time. In addition, it supports insertion (and deletion) of a substring of length y at an arbitrary position in T in $O(y + \sqrt{n})$ time. Later, Sahinalp and Vishkin [16] further improve the insertion and deletion time to $O(y + \log^3 n)$.

Note that the text T occupies only $O(n \log |\Sigma|)$ bits. Grossi and Vitter [9] recently proposed an index called Compressed Suffix Arrays (CSA) for a static text, which requires $O(\frac{1}{\epsilon}(nH_0 + n))$-bit space and supports pattern searching in $O(|P| \log^{1+\epsilon} n + occ \log^{\epsilon} n)$ time, where $0 < \epsilon \leq 1$ and $H_0 \leq \log |\Sigma|$ denotes the entropy of the text. Also, Ferragina and Manzini [8] have proposed another index called FM-index, which requires $O(nH_0 + n\frac{\log \log n}{\log^{\epsilon} n})$ bits and $O(|P| + occ \log^{\epsilon} n)$ searching time, for text with a small alphabet.

It was open whether there is a compressed index (i.e., using $O(n \log |\Sigma|)$ bits) that can manage a dynamic text efficiently. In this paper, we report the progress of this dynamic problem. Precisely, we propose an index that occupies $O(\frac{1}{\epsilon}(nH_0 + n))$ bits of space where $0 < \epsilon \leq 1$, while supporting pattern searching in $O(|P| \log^2 n(\log^{\epsilon} n + \log |\Sigma|) + occ \log^{1+\epsilon} n)$ time, and insertion/deletion of a substring of length y in $O((y+\sqrt{n}) \log^{2+\epsilon} n)$ amortized time.

Briefly speaking, our scheme is based on a reduction to the *dictionary* and *library management* problems (see e.g., [1, 2, 3, 7, 8, 14, 16]). In these two problems, we want to maintain a collection of texts $\{S_1, S_2, \ldots, S_k\}$ of total length n that allows efficient insertion and deletion of elements, and supports the following two queries:

- *dictionary matching*: Given a text T, find all occurrences of each of the S_i's in T.
- *library matching*: Given a pattern P, find all occurrences of P in each of the S_i's.

Ferragina and Manzini [8] have shown how to 'dynamize' the FM-index to give a compressed solution for the library management problem. CSA can also be adapted in a similar way. In this paper, we give a compressed solution for both the library management and the dictionary problem. Then, we make use of the interval partitioning technique in [6] to produce an indexing data structure for the compressed dynamic text problem.

The organization of this paper is as follows. In the next section, we review two compressed indexes for static text, which are the building blocks for our data structure. Section 3 describes a compressed index for the dictionary and library management problems, and Section 4 outlines the interval partition technique proposed in [6] for solving the dynamic text problem. Finally, we present our solution for the compressed dynamic text problem in Section 5.

IEEE COMPUTER SOCIETY

2 Compressed index for static texts

In this section, we review two compressed indexes for static texts, which are the building blocks of our compressed index for dynamic texts. Both of them occupies $O(n \log |\Sigma|)$ bits space, and support efficient pattern searching queries.

The first one is called Compressed Suffix Trees (CST) [11, 15]. A CST requires $O(nH_0 + n)$ bits of storage and can simulate a suffix tree [14] by supporting a set of tree navigation operations. In other words, any algorithm that is based on traversing a suffix tree can be solved using a CST, but with a blow up in running time. The performance of a CST is stated in the following lemma.

Let $T[1..n]$ be a text of length n, with characters drawn from an alphabet Σ.

Lemma 1. (Section 5 of [15]) *Let $0 < \epsilon \leq 1$. The CST of a text T can be constructed in $O(n \log^{1+\epsilon} n)$ time. The space occupancy is $O(\frac{1}{\epsilon}(nH_0 + n))$ bits. It supports searching of a pattern P in $O(|P| \log^{1+\epsilon} n + \log |\Sigma|) + occ \log^{1+\epsilon} n)$ time. In addition, it can simulate all constant time navigation operations on a suffix tree in $O(\log^{1+\epsilon} n + \log |\Sigma|)$ time.*

The second compressed indexing data structure is called Compressed Suffix Arrays (CSA) [9]. A CSA requires $O(\frac{1}{\epsilon}(nH_0 + n))$ bits of storage and can return the value of any entry in a suffix array in $O(\log^{\epsilon} n)$ time, where $0 < \epsilon \leq 1$. The performance of CSA is shown in the following lemma.

Lemma 2. (Lemma 9 of [12]) *Let $0 < \epsilon \leq 1$. The CSA of T can be constructed in $O(n \log \log |\Sigma|)$ time. The space occupancy is $O(\frac{1}{\epsilon}(nH_0 + n))$ bits. It supports searching of a pattern P in $O(|P| \log \log |\Sigma| + occ \log^{\epsilon} n)$ time, where occ denotes the number of occurrences of P in T.*

3 Compressed index for dictionary and library

In this section, we consider a compressed indexing data structure *CDL* for the dictionary and library management problems.

Consider a collection of texts $\{S_1, S_2, \ldots, S_k\}$. If $\Theta(n \log n)$ bits of storage are given, we can achieve optimal result by combining the data structures of [7] and [16]: insertion/deletion of an arbitrary text Y takes $O(|Y|)$ time, and answering the dictionary matching and the library matching queries takes $O(|T| + occ)$ time [16] and $O(|P| + occ')$ time [7], respectively, where occ and occ' denote the total number of occurrences reported in the the corresponding query.

If only $O(n \log |\Sigma|)$ bits of storage is allowed, any tree-based data structure would not have sufficient space for storing all the pointers, and updating would then become much more difficult. Ferragina and Manzini [8] have shown how to dynamize the FM-index to solve the library management problem. Their solution supports

IEEE COMPUTER SOCIETY

- finding all occurrences of P in the text collection in $O(|P| \log^3 n + occ' \log n)$ time in the worst case;
- insertion of a text Y in $O(|Y| \log n)$ amortized time, and deletion of Y in $O(|Y| \log^2 n)$ amortized time.

For the dictionary problem, as far as we know, no one has considered the setting under the $O(n \log |\Sigma|)$-bit storage requirement. In the following, we present a data structure which requires $O(\frac{1}{\epsilon} n \log |\Sigma|)$ bits where $0 < \epsilon \leq 1$, and which answers the dictionary matching and library matching queries in $O(|T| \log^2 n(\log^\epsilon n + \log |\Sigma|) + occ \log^{1+\epsilon} n)$ time and $O(|P| \log^2 n \log \log |\Sigma| + occ' \log^\epsilon n)$ time, respectively, in the worst case. For insertion or deletion of a text Y, it takes $O(|Y| \log^{2+\epsilon} n)$ amortized time. Our result makes use of CSA and CST instead of FM-index.

CST and static texts: We start by demonstrating the power of CST for indexing a *static* collection of texts. Firstly, with the various suffix tree navigation operations supported, it follows that CST can be directly plugged into any suffix-tree based scheme for representing a collection of texts, reducing the storage requirement. In particular, we can augment CST with a data structure of size $O(n)$ bits to support the dictionary matching query in $O(|T| \log n(\log^{1+\epsilon} n + \log |\Sigma|) + occ \log^{2+\epsilon} n)$ time, while using $O(\frac{1}{\epsilon} n \log |\Sigma|)$ bits space. (Details will be shown in the full paper.) Note that this result does not post any restriction on the texts S_i's in the collection. If we consider that no S_i is a proper substring of the other[1], we can use only the CST, and adapt the algorithm of Chang and Lawler [4] to simplify the searching method and reduce an $O(\log n)$ factor in the query time as follows.

Lemma 3. *Let $S = S_1\$S_2\$ \ldots S_k\$$ and assume that no S_i is a proper substring of the other. Given the CST of S, denoted by $CST(S)$, we can find all occurrences of each of the S_i's in a text T in $O(|T|(\log^{1+\epsilon} n + \log \Sigma) + occ \log^{1+\epsilon} n)$ time, where $0 < \epsilon \leq 1$.*

Proof. To report the occurrences, we traverse $CST(S)$ by matching characters of T, starting from the first character. This is similar to finding an occurrence of T in S using a suffix tree; but whenever we encounter a mismatch, or the previous step revealed an occurrence of S_i in T, we restrict the search by omitting the first character of the substring of T that is currently matched. For instance, suppose that we are at the position of the tree that matches the substring $T[q..r]$ (i.e., the position reached by matching $T[q..r]$ starting from the root of $CST(S)$). Then, we locate the position of the tree that matches the substring $T[q+1..r]$. This can be done by following a suffix link (which is a tree navigation operation) and then moving downwards a couple of nodes in the tree. Afterwards, we continue matching the next character $T[r+1]$ there.

Note that each S_i corresponds to a distinct leaf edge in $CST(S)$. At the r-th step where we match $T[r]$, we can check whether there is an occurrence of some S_i in T

[1] As to be shown in Section 5, the reduction from the compressed dynamic indexing problem involves a collection of strings satisfying this restriction.

IEEE COMPUTER SOCIETY

ending at $T[r]$, by examining whether matching an extra \$ would bring us to a leaf edge of some S_i.

Assuming that no S_i is a proper substring of the other, it is easy to verify that the above algorithm locates all occurrences of each S_i in T. For the running time, it follows since we can show that the number of tree navigation operations required (based on the arguments in [4]) is $O(|P|)$, and each operation requires at most $O(\log^{1+\epsilon}|S| + \log\Sigma)$ time by Lemma 1. This completes the proof of the lemma. □

On the other hand, CSA can be directly applied to solve the library management problem as follows.

Lemma 4. *Let $S = S_1\$S_2\$\ldots S_k$. Given the CSA of S, denoted by $CSA(S)$, we can find all occurrences of a pattern P in each of the S_i's in $O(|P|\log\log|\Sigma| + occ\log^{\epsilon} n)$ time.*

Proof. The lemma follows immediately from Lemma 2. □

CST, CSA, and dynamic texts: Next, we describe how to maintain a dynamic collection of texts based on the results on static texts. The idea follows closely to how Ferragina and Manzini 'dynamize' the FM-index, which is shown in [8]. Basically, we divide the texts in the collection into $O(\log n)$ groups. Groups are labeled by consecutive integers starting from 1, so that group i is either empty, or contains texts with total length in the range $[2^i, 2^{i+1})$. Note that each group is in fact a (sub-)collection of texts, and therefore we can support dictionary matching and library matching queries in each group using Lemmas 3 and 4. In other words, the total time for both queries will be increased by a factor of $O(\log n)$ in order to search all groups.

To insert a text X into the collection, we let $x = \lceil\log|X|\rceil$ and insert X into group x. However, there can be overflow during the insertion, since we require that the total length of texts in group x is less than 2^{x+1}. In this case, we merge group x and group $x+1$ (and possibly more groups) to form a new group, and construct a new CSA and CST accordingly. In the worst case, an insertion of a text can require a construction of CSA and CST for texts of total length $O(n)$; however, such a worst case would not be frequent, by considering that a particular text can be merged with the others at most $O(\log n)$ times. Thus, the amortized insertion time can be bounded by $O(\log n \cdot (|X|\log^{1+\epsilon} n))$.

For deletion, we apply a lazy update scheme so that the CST and the CSA of a group is re-constructed only when a fraction of $O(\frac{1}{\log n})$ of the total length is marked deleted. To do so, we keep a record on which texts are deleted, and a balanced search tree of height $O(\log n)$ for marking the intervals in the CSA and CST of the deleted texts. Then, deletion of S_i from the collection can be done in $O(\log n \cdot (|S_i|\log^{1+\epsilon} n))$ amortized time, but the time of the two queries is further increased by a factor of $O(\log n)$.

In summary, we have the following theorem.

IEEE COMPUTER SOCIETY

Theorem 1. *Consider all texts over an alphabet Σ. Let $\Delta = \{S_1, S_2, \ldots, S_k\}$ be a collection of texts with total length n, and let $0 < \epsilon \leq 1$. We can maintain a data structure CDL(S) of size $O(\frac{1}{\epsilon}(nH_0 + n))$ bits that supports*

- *finding all occurrences of S_i in a text T for all i in $O(|T| \log^2 n(\log^\epsilon n + \log|\Sigma|) + occ \log^{1+\epsilon} n)$ if no S_i is a proper substring of the other, where occ denotes the total number of occurrences of all the S_i's; in general, the time is $O(|T| \log^3 n(\log^\epsilon n + \log|\Sigma|) + occ \log^{2+\epsilon} n)$ if there is no restriction;*
- *finding all occurrences of a pattern P in the texts in Δ in $O(|P| \log^2 n \log\log|\Sigma| + occ' \log^\epsilon n)$ time, where occ$'$ denotes the total number of occurrences of P;*
- *inserting a text X into Δ in $O(|X| \log^{2+\epsilon} n)$ amortized time; and,*
- *deleting a text S_i from Δ in $O(|S_i| \log^{2+\epsilon} n)$ amortized time.*

The data structure in Theorem 1 can also be used to solve the *prefix-suffix* matching problem, which finds the longest suffix of a pattern P which is a prefix of any $S_i \in \Delta$. The result is stated in the following lemma.

Lemma 5. *The data structure in Theorem 1 supports finding the longest suffix of a pattern P which is a prefix of S_i for each i in $O(s_{max} \log n \log\log|\Sigma| + k \log^\epsilon n)$, where s_{max} denotes the length of the longest text in $\{S_1, S_2, \ldots, S_k\}$.*

Proof. To be given in the full paper. □

4 Interval partitioning and dynamic texts

After studying the dictionary and library problem, we are finally ready to discuss the indexing of a dynamic text to allow substring insertion and deletion at arbitrary positions. In this section we review a basic technique called interval partitioning, which was commonly used in the context of $\Theta(n \log n)$-bit index of a dynamic text (e.g., [5], [6]). The idea is that, given a text T, we divide it into "short", possibly overlapping *substrings*, so that a change in T at some position p is limited to changes in a few intervals around p. The drawback is that pattern searching would become a more involved operation.

The following interval partition scheme is proposed in [6]. Let ℓ be an integer (which will be chosen as $\sqrt{n}$). Suppose that T is partitioned into $k = \Theta(n/\ell)$ intervals $I_1, I_2, \ldots, I_k$ with each I_i is of length between ℓ to 2ℓ. Let $c_1 > c_2$ be some suitable constants. Define two boundary intervals I_0 and I_{k+1} containing $c_1\ell$ \$'s, where \$ is a new character not in Σ. To allow efficient searching, each interval I_i is to be represented by five "short" strings, which are of length $O(\ell)$:

- X_i to be the string $I_i \cdots I_{i+r}$, where r is the smallest integer such that $|I_i| + \cdots + |I_{i+r}| \geq c_1\ell$;

IEEE COMPUTER SOCIETY

- C_i and C'_i to be I_i's prefix and suffix of length ℓ, respectively;
- L_i to be the suffix of $I_0 I_i \cdots I_i$ of length $c_2\ell$ and R_i to be the prefix of $I_i \cdots I_k I_{k+1}$ of length $c_2\ell$.

In other words, T is represented by several collections of short strings (namely, X_i, C_i and C'_i, L_i, and R_i). Notice that for a pattern P of length at most $(c_1-2)\ell$, if it occurs in T, it must occur in some X_i. Thus, we only need to search the collection of X_i's.

Lemma 6. *Let* `Query-X(`P`)` *denote the query of finding all occurrences of a pattern P in each $X \in \{X_1, \cdots, X_k\}$. Suppose that* `Query-X(`$P$`)` *can be answered in t_X time. Then, if $|P| \leq (c_1-2)\ell$, all occurrences of P in T can be found in $O(t_X)$ time.*

Searching for longer patterns is non-trivial, yet Ferragina and Grossi [6] were able to devise an efficient algorithm based on the following batch queries to the four collections. The time complexity is stated in the following lemma.

- `Query-C(`P`)`: For each $X \in \{C_1, \cdots, C_k, C'_1, \cdots, C'_k\})$, find an occurrence of P.
- `Query-L(`P`)`: For each $X \in \{L_1, \cdots, L_k\}$, find P's *longest prefix* that is a suffix of X.
- `Query-R(`P`)`: For each $X \in \{R_1, \cdots, R_k\}$, find P's *longest suffix* that is a prefix of X.

Lemma 7. [6] *Suppose that* `Query-C(`P`)`, `Query-L(`P`)` *and* `Query-R(`P`)` *can be answered in t_C, t_L and t_R time, respectively. Then, if $|P| > (c_1-2)\ell$, all occurrences of P in T can be found in $O(t_C + t_L + t_R + occ)$ time, where occ denotes the number of occurrences of P in T.*

If $O(n \log n)$-bit space is given, one can use generalized suffix trees to represent the X_i's, C_i's, L_i's, and R_i's. Then `Query-X` can then be answered in $O(|P|+occ)$, where *occ* denotes the total number of occurrences of P in each X_i's, and the other three queries can be supported in $O(|P| + k + \ell)$ time.

In the next section, we propose a compressed data structure for storing the short strings in $O(n \log |\Sigma|)$ bits, and show how to support the queries efficiently. Then based on Lemma 6 and 7, any pattern can be searched efficiently.

5 Compressed index for dynamic texts

In this section, we describe the compressed indexing data structure for the *dynamic text* problem using $O(\frac{1}{\epsilon}(nH_0 + n))$ bits of space, where $0 < \epsilon \leq 1$.

Recall from the previous section that a text T is represented by several collections of short string (namely, X_i's, C_i's and C'_i's, L_i's and R_i's), each of length $O(\ell)$. We apply

IEEE COMPUTER SOCIETY

the compressed index discussed in Section 3 to represent each collection and support the update of strings in the collection, dictionary matching and library matching. Denote as CDL_X the resulting index $CDL(\{X_1, X_2, \ldots, X_k\})$, and similarly CDL_C, CDL_R, and CDL_L for respectively the indexes $CDL(\{C_1^l, \ldots, C_k^l, C_1^r, \ldots, C_k^r\})$, $CDL(\{R_1, \ldots, R_k\})$ and $CDL(\{L_1', \ldots, L_k'\})$, where L_i' denotes the reverse of L_i.

Note that all C_i's and C_i''s have equal length, so that none of them can be a proper substring of the other. This is also true for L_i's and R_i's. Thus, the time complexity stated in Theorem 1 for the dictionary matching query holds for CDL_C, CDL_R, and CDL_L. Also, recall that finding all occurrences of a pattern P in T can be reduced to answering some batch queries `Query-X`, `Query-C`, `Query-L` and `Query-R`. It is easy to see that these queries are readily supported by CDL_X, CDL_C, CDL_L and CDL_R based on Theorem 1 and Lemma 5. Precisely, `Query-X` corresponds to a library management query on CDL_X, `Query-C` corresponds to a dictionary matching query on CDL_C, while `Query-L` and `Query-R` correspond to prefix-suffix queries on CDL_L and CDL_R, respectively. The time complexity is stated in the following lemmas.

Lemma 8. *Given CDL_X, `Query-X(`P`)` can be answered in $O(|P| \log^2 n \log\log|\Sigma| + occ_X \log^\epsilon n)$ time, where occ_X denotes the total number of occurrence of P in all X_i's.*

Lemma 9. *Given CDL_C, `Query-C(`P`)` can be answered in $O(|P| \log^2 n(\log^\epsilon n + \log|\Sigma|) + k \log^{1+\epsilon} n)$ time.*

Proof. It follows from a minor modification to Lemma 3 which outputs only one occurrence for each of the S_i's in P. □

Lemma 10. *Given CDL_L and CDL_R, `Query-L(`P`)` and `Query-R(`P`)` can both be answered in $O(\ell \log n \log\log|\Sigma| + k \log^\epsilon n)$ time.*

By combining the above three lemmas with Lemma 6 and Lemma 7, we get the following result.

Corollary 1. *Let occ be the number of occurrences of P in T.*[2]

1. *If $|P| \leq (c_1 - 2)\ell$, all occurrences of P in T can be found in $O(|P| \log^2 n \log\log|\Sigma| + occ \log^\epsilon n)$ time;*
2. *otherwise, all occurrences can be found in $O(|P| \log^2 n(\log^\epsilon n + \log|\Sigma|) + k \log^{1+\epsilon} n + \ell \log n \log\log|\Sigma| + occ)$ time.*

Now, let us consider how to update the above *CDL*'s. Our method is based on the one used by [6] (where they are updating the generalized suffix trees instead). Firstly, suppose an arbitrary substring Y of length y is deleted from T. we observe that only those intervals $I_i, I_{i+1}, \ldots, I_j$ that overlap with Y is affected. (Note that the total

[2]Note that $occ_X \leq (c_1 - 2)occ$, where occ_X is the number of occurrences of P in all X_i's.

IEEE COMPUTER SOCIETY

length of these overlap intervals is at most $y + 4\ell$.) Consequently, we can update each of the *CDL*'s by removing those strings that depend on the overlap intervals from the corresponding collection. This in turn corresponds to the deletion of texts in CDL_X, CDL_C, CDL_L, and CDL_R of total length at most $c_1(y + 4\ell)$, $2(y + 4\ell)$, $c_2(y + 4\ell)$ and $c_2(y + 4\ell)$, respectively. To complete the discussion, note that there may be characters remaining in I_i and I_j after the deletion of Y. These characters can be merged together with I_{i-1} to form a new interval in T (and splitting may be required if the resulting length exceeds 2ℓ). Afterwards, we create new strings that depend on I_{i-1} and insert them into the corresponding *CDL*'s. Together, this corresponds to deletions and insertions of texts of total length $O(\ell)$ in each of the *CDL*'s.

On the other hand, when a text Y of length y is inserted into T, where the insertion point is either in the middle of some interval I_i, or between some interval I_i and I_{i+1}, we see that those strings in the *CDL*'s that depend on I_i (and I_{i+1} as well for the latter case) are needed to be removed. This corresponds to the deletions of texts of total length $O(\ell)$ in each of the *CDL*'s. Afterwards, we create intervals in T to accommodate Y. The total length of these intervals is at most $y + 2\ell$ (in case Y is inserted in the middle of I_i). This in turn corresponds to insertions of texts of total length $O(y + \ell)$ in each of the *CDL*'s.

In summary, an insertion or deletion of an arbitrary text of length y in T would require insertions and deletions of texts of total length at most $O(y + \ell)$ in CDL_X, CDL_C, CDL_L, and CDL_R. Thus, we have the following lemma.

Lemma 11. *We can update CDL_X, CDL_C, CDL_L and CDL_R in $O((y + \ell)\log^{2+\epsilon} n)$ amortized time, when an arbitrary text of length y is inserted or deleted in T.*

By setting $\ell = \sqrt{n}$, we have $k = O(\sqrt{n})$. This immediately leads to the following theorem.

Theorem 2. *Let $0 < \epsilon \leq 1$. We can maintain a compressed index for $T[1..n]$ in $O(\frac{1}{\epsilon}(nH_0 + n))$ bits that supports*

- *searching a pattern P in T, using $O(|P|\log^2 n(\log^{\epsilon} n + \log|\Sigma|) + occ\log^{1+\epsilon} n)$ time; and,*
- *insertion/deletion of an arbitrary text Y, using $O((|Y| + \sqrt{n})\log^{2+\epsilon} n)$ amortized time.*

Proof. The insertion and deletion time follows from Lemma 11. If $|P| \leq (c_1 - 2)\sqrt{n}$, the searching time follows, as it is $O(|P|\log^2 n \log\log|\Sigma| + occ\log^{\epsilon} n)$ by Corollary 1(1). Otherwise, we have $k = O(|P|)$, and the searching time follows from Corollary 1(2). □

References

[1] A. Aho and M. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

IEEE COMPUTER SOCIETY

[2] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic Dictionary Matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[3] A. Amir, M. Farach, R. Idury, A. La Poutre, and A. Schaffer. Improved Dynamic Dictionary Matching. *Information and Computation*, 119(2):258–282, 1995.

[4] W. I. Chang and E. L. Lawler. Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, 12(4/5):327–344, 1994.

[5] P. Ferragina and R. Grossi. Fast Incremental Text Editing. In *Proceedings of Symposium on Discrete Algorithms*, pages 531–540, 1995.

[6] P. Ferragina and R. Grossi. Optimal On-Line Search and Sublinear Time Update in String Matching. *SIAM Journal on Computing*, 27(3):713–736, 1998.

[7] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[8] P. Ferragine and G. Manzini. Opportunistic Data Structures with Applications. In *Proceedings of Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[9] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. In *Proceedings of Symposium on Theory of Computing*, pages 397–406, 2000.

[10] D. Gusfield, G. M. Landau, and B. Schieber. An Efficient Algorithm for All Pairs Suffix-Prefix Problem. *Information Processing Letters*, 41(4):181–185, 1992.

[11] W. K. Hon and K. Sadakane. Space-Economical Algorithms for Finding Maximal Unique Matches. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 144–152, 2002.

[12] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Sapce Barrier in Constructing Full-Text Indices. In *Proceedings of Symposium on Foundations of Computer Science*, pages 251–260, 2003.

[13] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[14] E. M. McCreight. A Space-economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[15] K. Sadakane. Compressed Suffix Tree with Full Functionality. Manuscript, 2003.

[16] S. C. Sahinalp and U. Vishkin. Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. In *Proceedings of Symposium on Foundations of Computer Science*, pages 320–328, 1996.

IEEE COMPUTER SOCIETY